CMPSCI 677 Operating Systems **Spring 2014**

Lecture 7: February 12

Lecturer: Prashant Shenoy *Scribe: Ashish Jain*

7.1 Communications in Distributed Systems

This lecture will deal with communication between different processes in distributed systems. Basic background of networking and RPCs will also be covered.
In a distributed system there are more than one component that are communicating over the network. Communication between any two components is inherent in any distributed system and the question is what abstraction the system should provide to enable this communication. At a very high level communication can be of two types

Unstructured Communication - It is essentially sharing memory or sharing data structures. Data can be put in shared buffer, another process can come and pick it up. In this manner process can interact with each other and pass messages between them. Application designer has to decide whether to use shared memory or shared data structures. No network support is required in such communications. But this is not a popular way of message passing in commercial systems.

Structured Communication - Most of the communication in distributed system is structured. These constitute of IPC's (Interprocess communications), RPC's or sending explicit message over sockets. Processes on same machine can use either of the above two methods. However, in Distributed Systems processes are on different machines. To communicate across machine, networking is required to pass messages. In this course mainly Structured Communication will be discussed.

7.1.1 Communication Protocols

This section gives a brief recap of the network protocol stack. Suppose an application on Machine A wants to communicate with an application on Machine B, the message from application needs to be processed through several layers of stack on Machine A, then go over a physical network and then reverse processing takes place in the protocol stack of Machine B, which then delivers the message to the application. Each layer has protocol which it follows to communicate with other layer in the network stack. Different layers in the network stack are discussed below :-

Physical Layer - It is essentially the lowest level of the stack (Ethernet, WiFi, 3G etc) which is the medium for communication between devices on network.

DataLink Layer - This layer is known as MAC protocol which is used to construct the packet in lowest layer and puts it out on the medium, essentially the ethernet protocol runs on this layer. Data packets are encoded and decoded into bits. It furnishes transmission protocol knowledge and management and handles errors in the physical layer, flow control and frame synchronization.

Network Layer - Routing and forwarding are functions of this layer, as well as *addressing, internetworking, error handling, congestion control and packet sequencing.* This layer is responsible for hop by hop

communication. A data packet sent out by machine A might have to traverse through multiple hops before it reaches its final destination. When packet is delivered at one hop it decides what next has to be taken at this point until it reaches its final destination.

Transport Layer - This layer is responsible for end to end communication. This layer provides transparent transfer of data between end systems, or hosts, and is responsible for end-to-end error recovery and flow control (e.g. rate of transmission etc.). It ensures complete data transfer.

Application Layer - This layer supports application and end-user processes. This layer provides application services for file transfers, e-mail, and other network software services. Telnet, HTTP and FTP are applications that exist entirely in the application level.

7.1.2 Layered Protocols

The actual message which needs to be sent to a destination is called *Payload*, as the message traverses down the network stack each layer processes the message and adds a headers to the message packet. For example, *http* is an application layer protocol, thus http header may be added in front of the payload. In Transport Layer, TCP will add a header to the payload. In Network Layer, IP will add a header which denotes the source and destination addresses of the machines which is used at each hop to identity where the packet came from and where it is headed to according to which routing of that packet is done. In DataLink layer, suppose ethernet is being used to transmit the message over network, then a ethernet header is added to the payload. At the trailer end, there can be a checksum which is used to check for errors in transmission. At the receiving end, each layer analyses the header and then strips it off and hands it to the layer above it. Receiver application layer receives the original message passed from sender application layer.

7.1.3 Middleware Protocols

In Distributed Systems environment, the middleware layer sits in between Application Layer and the Transport Layer or the Operating System. The middleware protocol provides higher level service to application which is even much more advanced than session and presentation layer. For example, Multicast, Broadcast, Unicast services are provided by the middleware. Application just needs to send a packet for broadcast to the middleware and then later takes care of sending the message to the network. Essentially middleware abstracts the Application from the TCP / IP and other networking aspects of the distributed system. This will be discussed more in upcoming classes.

7.1.4 Client-Server TCP

In a typical client server architecture, client constructs a request and sends it over to the server, server processes the request at its end and sends back the response to the client. The diagram in the slides show how message passing is done using TCP / IP. For a client to send a message to the server, the client has to establish a TCP connection to the server first, i.e. a 3-way handshake is required to establish a TCP connection. The client sends a SYN packet to the server, which means that the client wants to establish a TCP connection with the server. Server in turn sends back a SYN and a ACK(SYN) packet back to the client. ACK(SYN) is the acknowledgement sent from the server to the client and SYN is sent to the client to denote that server wants to setup a connection with it. The client then acknowledges the SYN request of server by sending a ACK(SYN) message back to the server. Thus this is a 3-way handshake mechanism to establish a TCP connection. Only after this TCP connection have been setup, the client can send a request message to the server. One the client sends the request to the server, it send a message FIN to the server denoting that the client has finished sending the message to the server. Now server processes the request

and while it is processing the request it sends back a ACK(req + FIN) to denote that it has received the request and finish message. Once the request is processed in the server the answer is send back to the client and then a FIN message is sent to the client to denote end of reply. The client then acknowledges back to the server by sending ACK(FIN) message that it has received the answer. The last 3 messages were to tear down the TCP connection. Thus we see that in total 9 packets were sent for the whole process which in turn results in overhead.

To overcome this overhead, Client-Server TCP is used to optimize the entire process.Essentially in this the messages are batched together to reduce the overhead. The first packet sent from client is (SYN, request, FIN) , which means, that client wants to setup TCP connection with the server, the request is embedded in the same packet and then the client says it is done sending the message. The server then processes the request and sends back SYN,ACK(FIN),answer, FIN which means that it wants to setup a connection with the client, then acknowledge clients Finish request, answer to the request and then finish message from the server side. Then client just send ACK(FIN) which means that it has received the answer from the server and TCP connection can be torn down now. This architecture was merely a proposal but in real it is seldom used. This architecture is most useful for sending one message and receiving one reply from server. Otherwise in general, the 1st method of sending messages is used.

7.1.5 Push or Pull Architecture

- *Client - Pull Architecture* - The architecture that we discussed in the previous section is client pull architecture. The client sends in request to the server to *pull* data from it. For example, HTTP is a client pull architecture protocol where a client sends a webpage request to get a reply from the web server. In this type of architecture the servers are *stateless* , i.e. servers don't need to keep track of the clients its communicating with. If the client wants to get some more data, it will send a new request to the server, but server doesn't need to worry about it. From a failure prospective this architecture is more robust since no state of clients is maintained, thus if the server crashes, the client just needs to send a new request for the data. This architecture is more scalable from memory point of view since states of client need not be maintained on the server. But this architecture is not efficient from communications point of view. To get one piece of data two messages are exchanged i.e. a request and a reply, which results in a overhead.

- *Server - Push Architecture* - In this architecture, the client may not send explicit requests for data, server asynchronously pushes data to the client. For example, in Video and Audio Streaming, the client just sends a one time request for the data on the server, after that the server starts pushing data to the client at certain intervals, the client needn't send explicit requests for different pieces of video file from the server. The client just listens to the socket and data keeps coming in from the server. In this type of architecture the servers are *stateful*, i.e. the servers keep tracks of the clients its communicating with. The video can be a lengthy clip and server needs to know to which client it has to push the data and at what intervals. This architecture is not robust from failure point of view. If the server reboots, then all the state of clients is lost from memory and the server will lose track of the data it was streaming to different clients, thus everything will have to done from the start. This architecture is less scalable from memory point of view since states of clients needs to be maintained in the servers memory. But from communication point of view this architecture is more efficient, since client just sends in one request and server starts pushing data to the client which has less overhead than client - pull architecture.

  Thus during the design of Distributed System architecture we need to keep in mind whether we want a Push based or Pull based architecture.

7.1.6 Group Communication

This is also referred to as broadcast or multicast where the same message is sent to more than one recipient. For example, sending an email to a mailing list, where in one email is sent to multiple users at once who have subscribed to that mailing list. Same can be done for network messages, clients can subscribe to a group and anyone who posts a message to that group will be sent to all the clients automatically. For example, audio broadcast, where lot of users have tuned in to listen to it, in this case server / sender is not sending the message to each of the users separately, it sends a message to the middleware and this middleware then sends the message to users in the group.

7.1.7 Remote Procedure Calls

The goal of Remote Procedure Calls(RPCs) is to make distributed computing look like centralized computing. It allows remote services to be called as procedures. Rather than sending abstract messages between a client and a server and letting the server process the message and send back the response, the idea was to directly let the client call functions on the server. This makes things easier for the programmer as now he/she just have to call a remote function on the server instead of sending explicit message to the server. Example of RPC implementation is given on the lecture slides. The server has a function $a$dd() which accepts two integers as addends. The client calls this remote function on the server and passes two two integers which needs to be added. When client calls this add() function the RPC runtime system internally sends a request to the server, which in turn returns the value. The RPC takes care of passing the variables to the server, constructing a message to call the function, TCP/IP connection etc, the programmer needn't deal with all this complexity, he/she just need call the function as if it was a local function call. C doesn't have support for RPC. It has library which provides API to use RPCs. Java has inbuilt RPC support.

7.1.8 RPC Parameter Passing

There are some caveats when calling Remote Procedures. While calling Remote procedures, the user can't pass parameter by reference, i.e. a pointer to an address space can't be passed to the server because when it is dereferenced, it will not have the same value which the address space on client was holding. Thus the user needs to pass parameter by value. Likewise, global variables can't be used for RPC.

The system generates stubs, these are essentially proxies at both ends that system generates for the application that deals with setting up a network connection to the server, constructing a message whenever a remote call is made, sending the message, wait for reply, all of this is taken care by these stubs. All of this is abstracted from the programmer with the help of stubs.

Flattening and Marshaling of arguments, it takes care of the arguments passed to the function call, whether it be a structure or an array, whether it is calling the function on the server with different OS or hardware architecture (little endian or big endian), everything is taken care by the stubs which converts native data format into universally agreed format, external data representation (XDR) thus the programmer needn't worry about all these complications.

7.1.9 Client and Server Stubs

- Client makes procedure call (just like a local procedure call) to the client stub.
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages

- Packaging parameters is called marshalling
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)

7.1.10 Binder : Port Mapper

When the client and the server startup, and the client invokes a message at the server, it needs to first find the server. This is done through Port Mapper, when the server starts up it registers itself at the port mapper with the list of API's that it exposes and the port that it is listening on. Therefore when the client starts up, it first checks with the port mapper for the server with the remote procedure that the client wants to execute. From the port mapper it gets the server's port number, this is a one time setup. Once this is done, the stub takes over and performs required operations. Also it is assumed that the IP address of the server is known but which port it is listening on is unknown, which is then found through port mapper.

7.1.11 Failure Semantics

RPC can run over TCP (reliable) or UDP (unreliable). What happens if a message client sends to server gets lost? There are two answers to this question. If you use RPC over reliable transport protocol (TCP), it will retransmit the message in case it is lost. RPC doesn't have to care about message lost scenario. Original version of RPCs ran on UDP because they didn't want overhead cost of TCP message transmission (9 message for single request and response). In case of Local area network this option is fine because package lost chances are very low. In UDP, if package is lost it is RPCs runtime job to deliver the packet.

7.1.12 Various kind of failure

- No server present to register - There is nothing you can do. Error will be returned.
- Lost request messages - You will use timeout mechanism. If reply is not received, you will reset the request. Same TCP has done for you, now you do at RPC runtime system.
- Lost replies - In case of TCP it will retransmit the answer. However, in case of UDP, a simple scenario to consider is to resend the request. But if the request changes the state on server, it might create problems. For example, if a request increases the counter on server, then resending the request will make counter twice, which is not desired. Consider a scenario where you want to make flight booking. You send a request to book a seat and server books it for you. In this case if reply is lost and you again send the request to book a seat, then you end up booking two seats. To handle these problems, we need to make sure that operations are idempotent i.e. end result doesn't depend on request. To implement it, you buffer the result and retransmit it if reply is lost. A unique id is assigned to each request and replies are handled accordingly. Therefore, you won't process the request again in case reply is already present in the buffer.
- Server Failure - You have to send request again if server goes down. There are various semantics associated to it. In At least once semantics, it can cause problems if it is not idempotent operation because you might end up executing request several times to get a response. Most desirable is Exactly once semantic. Whenever reply comes back, you are sure that request has been executed once. It is hard to achieve because you have to do extra bookeeping. In case of At most one semantics you might have no response which means you receive error on sending request.